

CHAPTER 8



MongoDB Explained

*“MongoDB explained covers **deep-dive concepts** of what happens under the hood in MongoDB.”*

In this chapter, you will learn how data is stored in MongoDB and how writes happen using journaling. Finally, you will learn about GridFS and the different types of indexes available in MongoDB.

Data Storage Engine

In the previous chapter, you looked at the **core services** that are deployed as part of MongoDB; you also looked at **replica sets** and **sharding**. In this section, we will talk about the **data storage engine**.

MongoDB uses **MMAP** as its **default storage engine**. This engine works with memory-mapped files. Memory-mapped files are data files that are placed by the operating system in memory using the `mmap()` system call. `mmap` is a feature of OS that maps a file on the disk into **virtual memory**.

Virtual memory is not equivalent to physical memory. Virtual memory is space on the computer's hard disk that is used in conjunction with physical RAM.

MongoDB uses memory-mapped files for any data interaction or data management activity. As and when the documents are accessed, the data files are memory mapped to the memory. MongoDB allows the OS to control the memory mapping and allocate the maximum amount of RAM. Doing this results in minimal effort and coding at MongoDB level. The caching is done based on LRU behavior wherein the least recently used files are moved out to disk from the working set, making space for the new recently and frequently used pages.

However, this method comes with its own drawbacks. For instance, MongoDB has no control over what data to keep in memory and what to remove. So every server restart will lead to a page fault because every page that is accessed will not be available in the working set, leading to a long data retrieval time.

MongoDB also has no control over prioritizing the content of the memory. Given an evacuation situation, it can mention what content needs to be maintained in the cache and what can be removed. For example, if a read is fired on a large collection that is not indexed, it might result in loading the entire collection to memory, which might lead to evacuation of the RAM contents including removal of indexes of other collections that might be very important. This lack of control might also result in shrinking the cache assigned to MongoDB when any external process outside MongoDB tries to access a large portion of memory; this eventually will lead to slowness in the MongoDB response.

With the release of version 3.0, MongoDB comes along with a pluggable storage engine API wherein it enables you to select between the storage engines based on the workload, application need, and available infrastructure.

The vision behind the pluggable storage engine layer is to have one data model, one querying language, and one set of operational concerns, but under the hood many storage engine options optimized for different use cases, as shown in Figure 8-1.

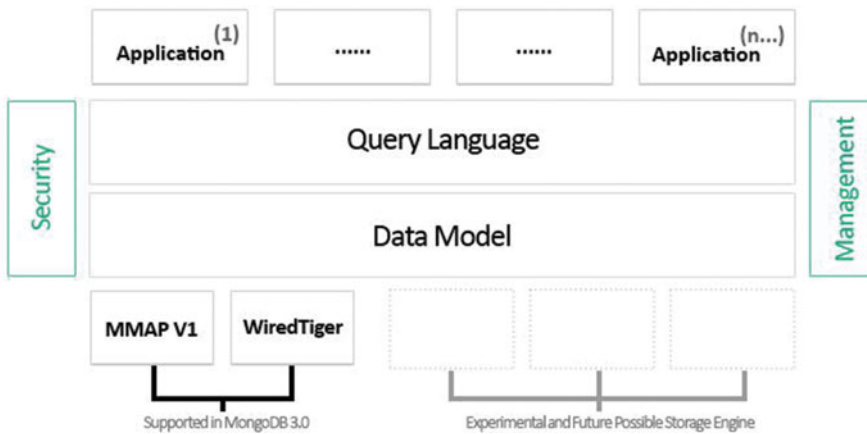


Figure 8-1. Pluggable storage engine API

The pluggable storage engine feature also provides flexibility in terms of deployments wherein multiple types of storage engines can coexist in the same deployment.

MongoDB version 3.0 ships with two storage engines.

The default, **MMAPv1**, is an improved version of the MMAP engine used in the prior versions. The updated MongoDB MMAPv1 storage engine implements collection-level concurrency control. This storage engine excels at workloads with high volume reads, inserts, and in-place updates.

The new WiredTiger storage engine was developed by the architects of Berkeley DB, the most widely deployed embedded data management software in the world. WiredTiger scales on modern multi-CPU architectures. It is designed to take advantage of modern hardware with multi-core CPUs and more RAM.

WiredTiger stores data in compressed format on the disk. Compression reduces the data size by up to 70% (disk only) and index size by up to 50% (disk and memory both) depending on the compression algorithm used. In addition to reduced storage space, compression enables much higher I/O scalability as fewer bits are read from disk. It provides significant benefits in the areas of greater hardware utilization, lower storage costs, and more predictable performance.

The following compression algorithms are available to choose from:

- Snappy is the default, which is used for documents and journals. It provides a good compression ratio with little CPU overhead. Depending on data types, the compression ratio is somewhere around 70%.
- zlib provides extremely good compression but at the expense of extra CPU overhead.
- Prefix compression is the default used for indexes, reducing the in-memory footprint of index storage by around 50% (workload dependent) and freeing up more of the working set for frequently accessed documents.

Administrators can modify the default compression settings for all collections and indexes. Compression is also configurable on a per-collection and per-index basis during collection and index creation.

WiredTiger also provides granular document-level concurrency. Writes are no longer blocked by other writes unless they are accessing the same document. Thus it supports concurrent access by readers and writers to the documents in a collection. Clients can read documents while write operations are in progress, and multiple threads can modify different documents in a collection at the same time. Thus it excels for write-intensive workloads (7-10X improvement in write performance).

Higher concurrency also drives infrastructure simplification. Applications can fully utilize available server resources, simplifying the architecture needed to meet performance SLAs. With the more coarse grained database-level locking of previous MongoDB generations, users often had to implement sharding in order to scale workloads stalled by a single write lock to the database, even when sufficient memory, I/O bandwidth, and disk capacity was still available in the host system. Greater system utilization enabled by fine-grained concurrency reduces this overhead, eliminating unnecessary cost and management load.

This storage engine provides control to you per collection per index level to decide on what to compress and what not to compress.

The WiredTiger storage engine is only available with 64-bit MongoDB.

WiredTiger manages data through its cache. The WiredTiger storage engine gives more control of memory by allowing you to configure how much RAM to allocate to the WiredTiger cache, defaulting to either 1GB or 50% of available memory, whichever is larger.

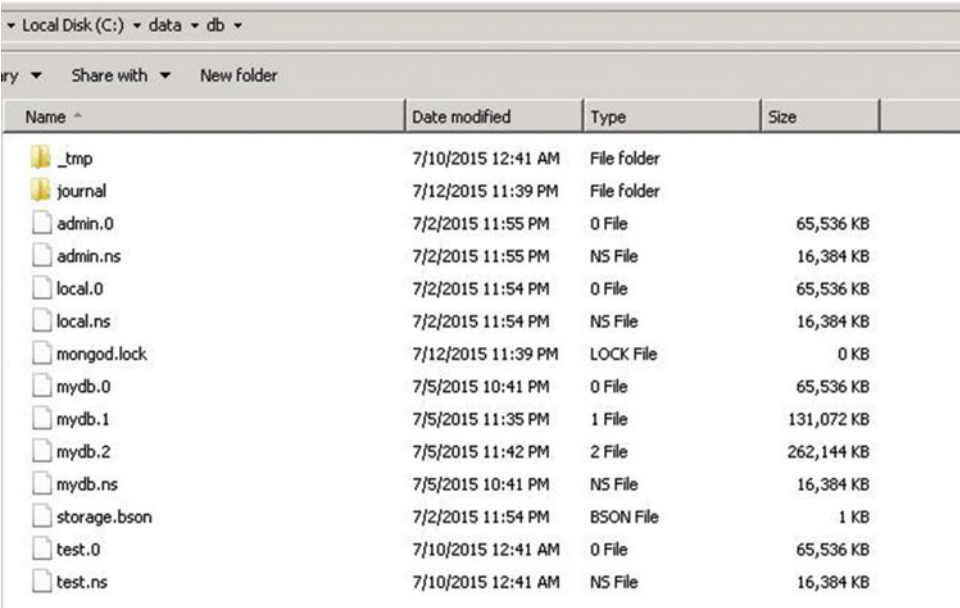
You will next briefly look at how the data is stored on the disk.

Data File (Relevant for MMAPv1)

First, let's examine the **data file**. As you have seen, under the core services the default data directory used by mongod is `/data/db/`.

Under this directory there are **separate files** for every database. Each database has a single **.ns** file and multiple data files with monotonically increasing numeric extensions.

For example, if you create a database called **mydbpoc**, it will be stored in the following files: **mydb.ns**, **mydb.1**, **mydb.2**, and so on, as shown in Figure 8-2.



Name	Date modified	Type	Size
_tmp	7/10/2015 12:41 AM	File folder	
journal	7/12/2015 11:39 PM	File folder	
admin.0	7/2/2015 11:55 PM	0 File	65,536 KB
admin.ns	7/2/2015 11:55 PM	NS File	16,384 KB
local.0	7/2/2015 11:54 PM	0 File	65,536 KB
local.ns	7/2/2015 11:54 PM	NS File	16,384 KB
mongod.lock	7/12/2015 11:39 PM	LOCK File	0 KB
mydb.0	7/5/2015 10:41 PM	0 File	65,536 KB
mydb.1	7/5/2015 11:35 PM	1 File	131,072 KB
mydb.2	7/5/2015 11:42 PM	2 File	262,144 KB
mydb.ns	7/5/2015 10:41 PM	NS File	16,384 KB
storage.bson	7/2/2015 11:54 PM	BSON File	1 KB
test.0	7/10/2015 12:41 AM	0 File	65,536 KB
test.ns	7/10/2015 12:41 AM	NS File	16,384 KB

Figure 8-2. Data files

For each new numeric data file for a database, the size will be double the size of the previous number data file. The limit of the file size is 2GB. If the file size has reached 2GB, all subsequent numbered files will remain 2GB in size. This behavior is by design. *This behavior ensures that small databases do not waste too much space on disk, and large databases are mostly kept in contiguous regions on the disk.*

Note that in order to ensure consistent performance, MongoDB preallocates data files. The preallocation happens in the background and is initiated every time a data file is filled. This means that the MongoDB server always attempts to keep an extra, empty data file for every database in order to avoid blocking on file allocation.

If multiple small databases exist on disk, using the `storage.mmapv1.smallFiles` option will reduce the size of these files.

Next, you will see how the data is actually stored under the hood. Doubly linked lists are the key data structure used for storing the data.

Namespace (.ns File)

Within the data files you have data space divided into **namespaces**, where the namespace can correspond to either a **collection** or an **index**.

The metadata of these namespaces are stored in the **.ns file**. If you check your data directory, you will find a file named **[dbname].ns**.

The size of the **.ns** file that is used for storing the metadata is **16MB**. This file can be thought of as a big hash table that is partitioned into small buckets, which are approximately **1KB** in size.

Each bucket stores metadata specific to a **namespace** (Figure 8-3).



Figure 8-3. Namespace data structure

Collection Namespace

As shown in Figure 8-4, the collection namespace bucket contains metadata such as

- Name of the collection
- A few statistics on the collection such as count, size, etc. (This is why whenever a count is issued against the collection it returns quick response.)
- Index details, so it can maintain links to each index created
- A deleted list
- A doubly linked list storing the extent details (it stores pointer to the first and the last extent)

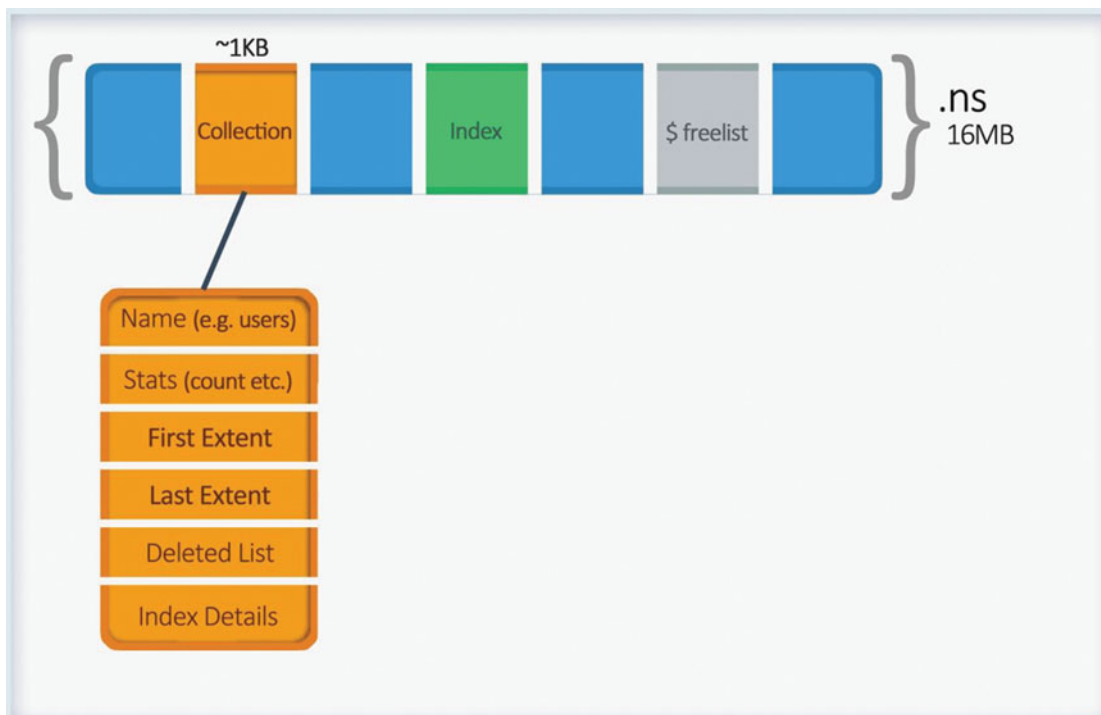


Figure 8-4. Collection namespace details

Extent

Extent refers to a **group of data records** within a data file, so a group of extents forms the complete data for a namespace. An extent uses **disk locations** to refer to the location on the disk where the data is **actually residing**. It consists of two parts: **file number** and **offset**.

The file number specifies the data file it's pointing to (0, 1, etc.).

Offset is the position within the file (how deep within the file you need to look for the data). The offset size is 4KB. Hence the offset's maximum value can be up to $2^{31}-1$, which is the maximum file size the data files can grow to (2048MB or 2 GB).

As shown in Figure 8-5, an **extent data structure** consists of the **following** things:

- **Location** on the disk, which is the file number it is pointing to.
- Since an extent is stored as a doubly linked list element, it has a pointer to the next and the previous extent.
- Once it has the file number it's referring to, the group of the data records within the file it's pointing to are further stored as **doubly linked list**. Hence it maintains a pointer to the first data record and the last data record of the data block it's pointing to, which are nothing but the offsets within the file (how deep within the file the data is stored).

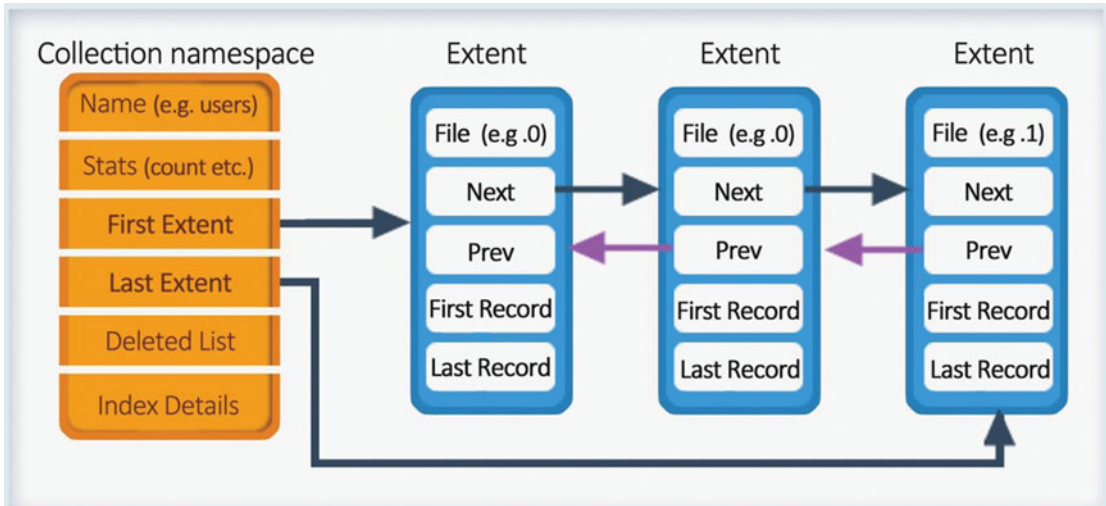


Figure 8-5. Extent

Data Record

Next you will look at the **data record structure**. The data structure consists of the following details:

- Since the data record structure is an element of the extent's doubly linked list, it stores information of the **previous and the next record**.
- It has **length** with **headers**.
- The **data block**.

The data block can have either a BTree Bucket (in case of an index namespace) or a BSON object. You will be looking into the BTree structure in a while.

The BSON object corresponds to the actual data for the collection. The size of the BSON object need not be same as the data block. Power of 2-sized allocation is used by default so that every document is stored in a space that contains the document plus extra space or padding. This design decision is useful to avoid movement of an object from one block to another whenever an update leads to a change in the object size.

MongoDB supports multiple allocation strategies, which determine how to add padding to a document (Figure 8-6). As in-place updates are more efficient than relocations, all padding strategies trade extra space for increased efficiency and decreased fragmentation. Different workloads are supported by different strategies. For instance, exact fit allocation is ideal for collections with insert-only workloads where the size is fixed and never varies, whereas power of 2 allocations are efficient for insert/update/delete workloads.

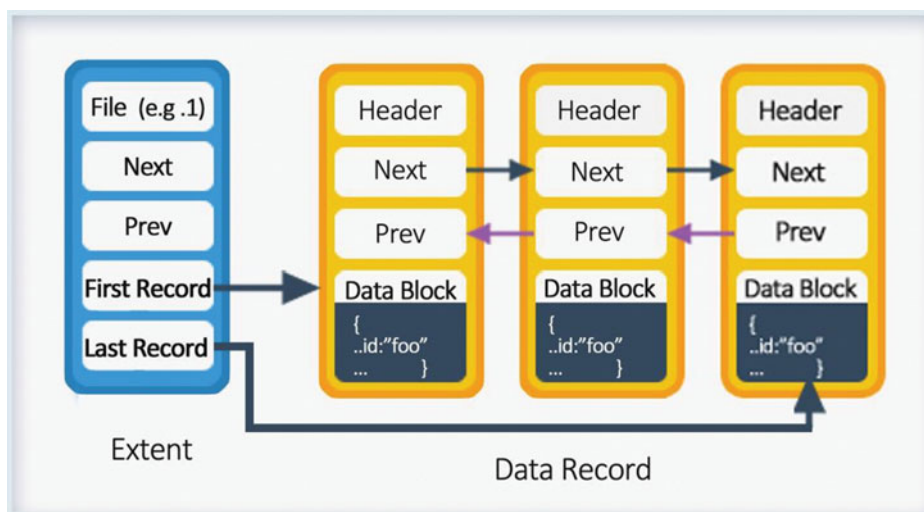


Figure 8-6. Record data structure

Deleted List

The deleted list stores details of the **extent** whose data has been **deleted** or **moved** (movement whenever an update has caused change in size, leading to non-fitment of data in the allocated space).

The size of the record determines the bucket in which the free extent needs to be placed. Basically these are bucketed single linked lists. When a new extent is required for fitting the data for the namespace, it will first search the free list to check whether any appropriate size extent is available.

In Summary

Hence you can assume the data files (files with numeric extensions) to be divided across different collection namespaces where **extents** of the namespace specify the **range of data** from the data file belonging to that respective collection.

Having understood how the data is stored, now let's see how `db.users.find()` works.

It will first check the `mydb.poc.ns` file to reach the users' namespace and find out the first extent it's pointing to. It'll follow the first extent link to the first record, and following the next record pointer, it will read the data records of the first extent until the last record is reached. Then it will follow the next extent pointer to read its data records in a similar fashion. This pattern is followed until the last extent data records are read.

\$freelist

The .ns file has a special **namespace** called \$freelist for extents. \$freelist keeps track of the extents that are **no longer used**, such as extents of a dropped index or collection.

Indexes BTree

Now let's look at how the **indexes are stored**. The BTree structure is used for storing the indexes. A BTree is shown in Figure 8-7.

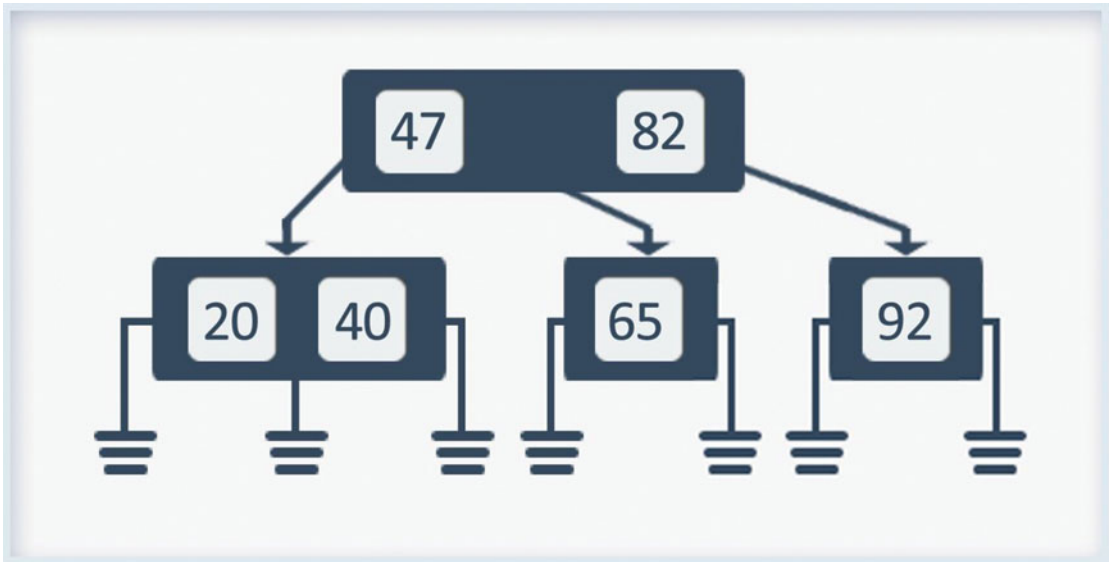


Figure 8-7. BTree

In a standard implementation of BTree, whenever a **new key** is inserted in a BTree, the default behavior is as shown in Figure 8-8.

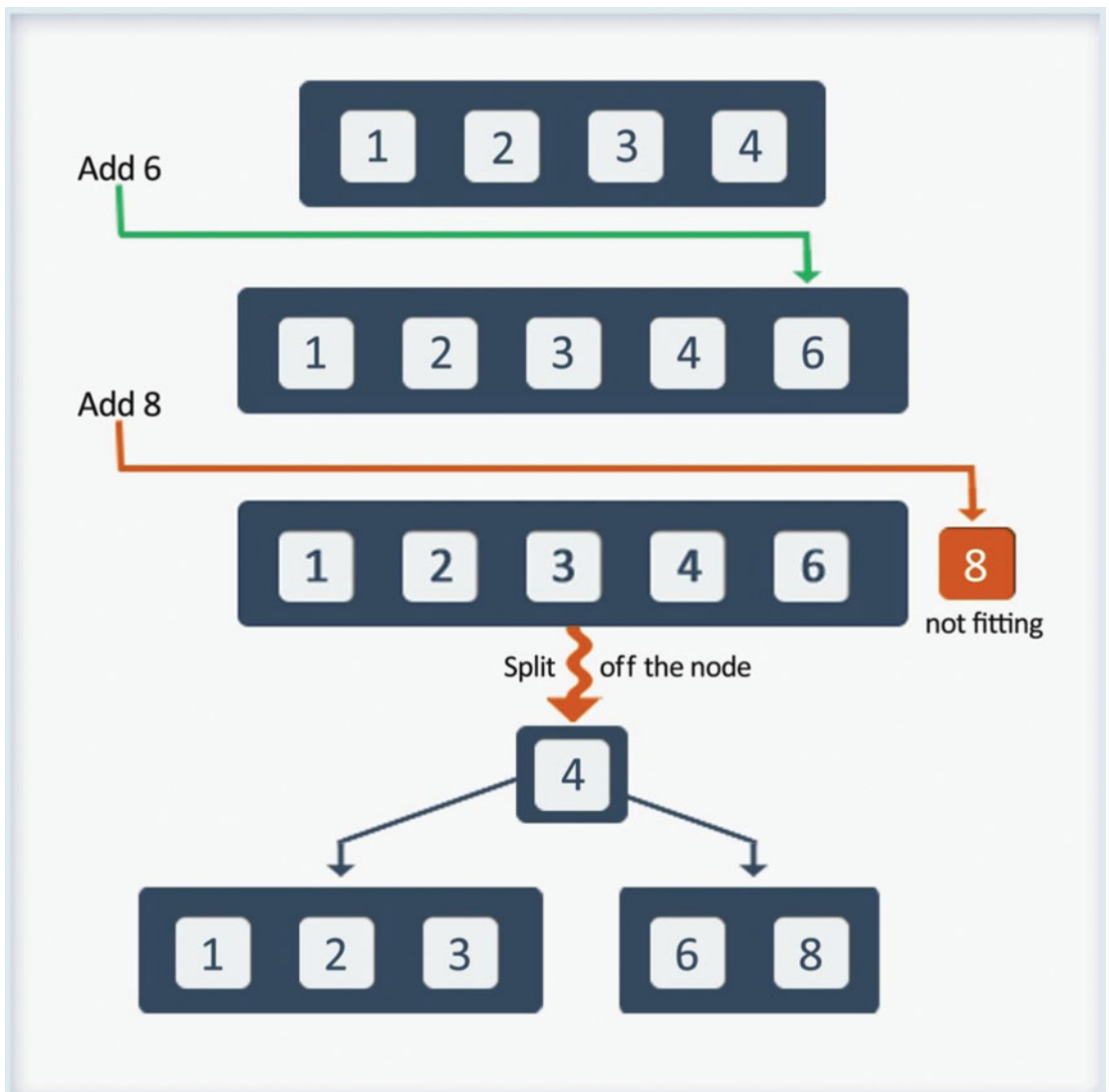


Figure 8-8. B-Tree standard implementation

There's a slight variation in the way MongoDB implements the BTree. In the above scenario, if you have keys such as Timestamp, ObjectID, or an incrementing number, then the buckets will always be half full, leading to lots of wasted space. In order to overcome this, MongoDB has modified this slightly. Whenever it identifies that the index key is an incrementing key, rather than doing a 50/50 split, it does a 90/10 split as shown in Figure 8-9.

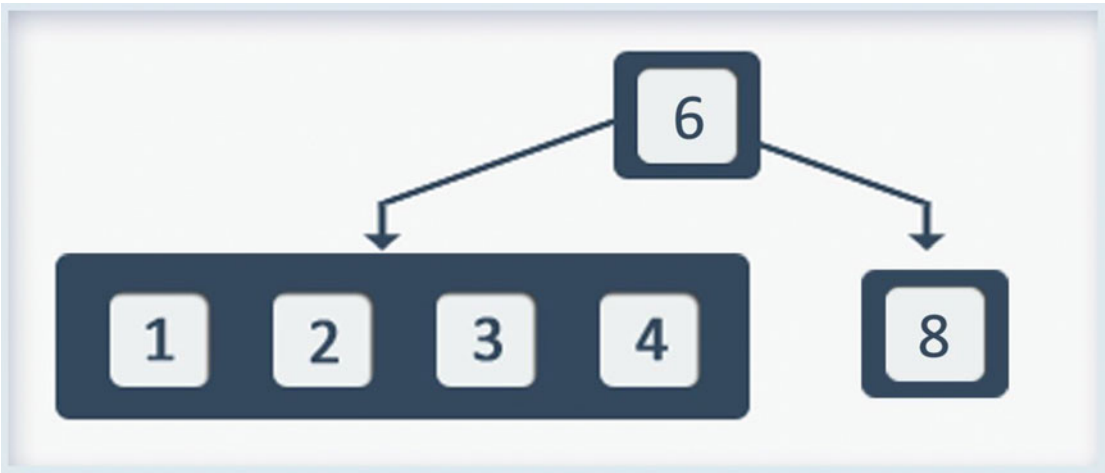


Figure 8-9. MongoDB's B-Tree 90/10 split

Figure 8-10 shows the bucket structure. Each bucket of the BTree is of 8KB.

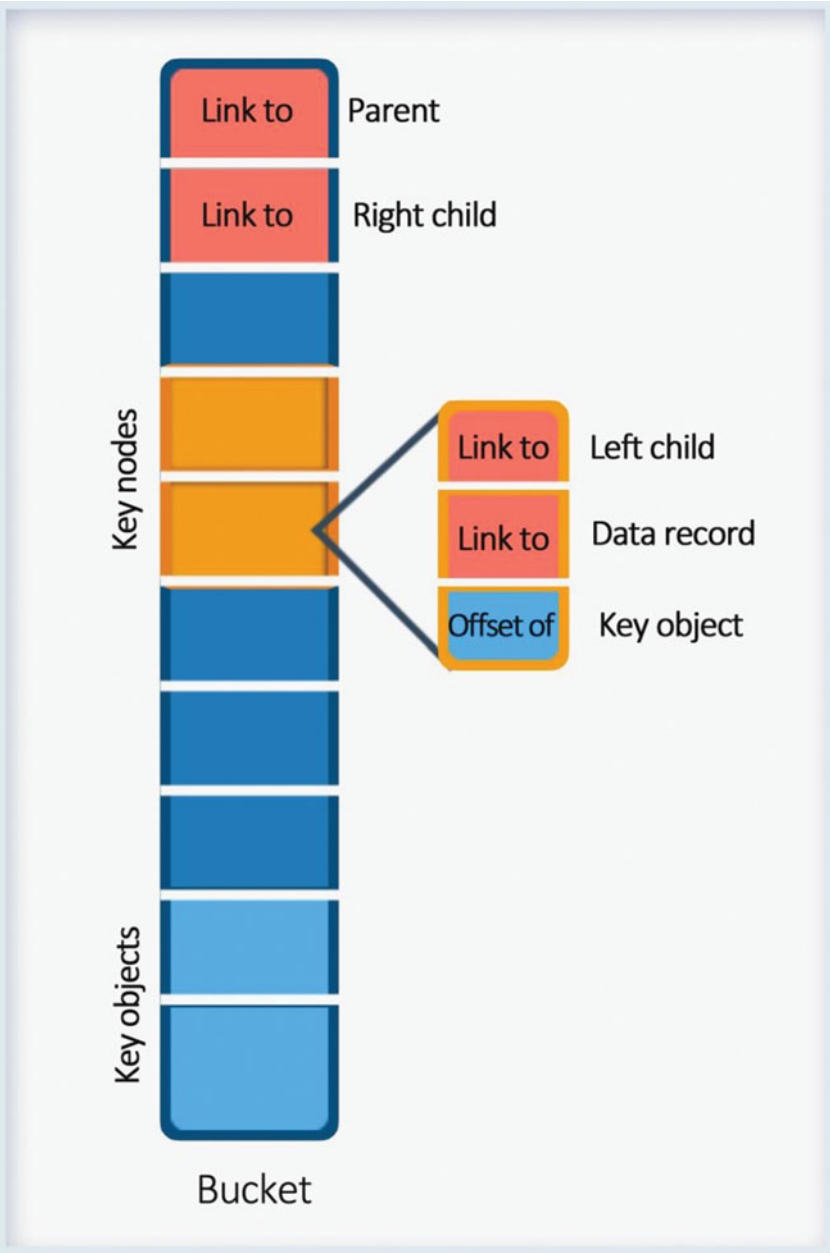


Figure 8-10. BTree bucket data structure

The bucket consists of the following:

- Pointer to the parent
- Pointer to the right child
- Pointer to key nodes
- A list of key objects (these objects are of varying size and are stored in an unsorted manner; these objects are actually the value of the index keys)

Key Nodes

Key nodes are **nodes of a fixed size** and are **stored in a sorted manner**. They enable easy split and movement of the elements between different nodes of the **BTree**.

A key node contains the following:

- A **pointer** to the **left child**
- The **data record** the index key belongs to
- A key offset (the offset of the key objects, which basically tells **where in the bucket** the key's value is stored)

Data File (Relevant for WiredTiger)

In this section, you will look at the content of data directory when the mongod is started with the **WiredTiger storage engine**.

When the storage option selected is WiredTiger, **data, journals, and indexes** are **compressed on disk**. The compression is done based on the compression algorithm specified when **starting the mongod**. **Snappy** is the **default compression option**.

Under the data directory, there are separate compressed **wt files** corresponding to each **collection** and **indexes**. Journals have their **own folder** under the data directory.

The compressed files are actually created when data is inserted in the collection (the files are allocated at write time, no preallocation).

For example, if you create collection `collection-0-2259994602858926461` files and the associated index will be stored in `index-1-2259994602858926461`, `index-2-2259994602858926461`, and so on.

In addition to the collection and index compressed files, there is a `_mdb_catalog` file that stores **metadata mapping collection** and **indexes** to the files in the data directory. In the above example it will store mapping of collection users to the wt file `collection-0-2259994602858926461`. See Figure 8-11.

Name	Date modified	Type	Size
journal	7/19/2015 10:12 PM	File folder	
_mdb_catalog	7/19/2015 10:46 PM	WT File	36 KB
collection-0--2259994602858926461	7/19/2015 10:13 PM	WT File	16 KB
collection-2--2259994602858926461	7/19/2015 10:13 PM	WT File	16 KB
collection-4--2259994602858926461	7/19/2015 10:24 PM	WT File	16 KB
collection-6--2259994602858926461	7/19/2015 10:24 PM	WT File	16 KB
collection-8--2259994602858926461	7/19/2015 10:26 PM	WT File	16 KB
collection-10--2259994602858926461	7/19/2015 10:27 PM	WT File	16 KB
collection-18--2259994602858926461	7/19/2015 10:39 PM	WT File	492 KB
index-1--2259994602858926461	7/19/2015 10:13 PM	WT File	16 KB
index-3--2259994602858926461	7/19/2015 10:13 PM	WT File	16 KB
index-5--2259994602858926461	7/19/2015 10:24 PM	WT File	16 KB
index-7--2259994602858926461	7/19/2015 10:24 PM	WT File	16 KB
index-9--2259994602858926461	7/19/2015 10:26 PM	WT File	16 KB
index-11--2259994602858926461	7/19/2015 10:27 PM	WT File	16 KB
index-19--2259994602858926461	7/19/2015 10:39 PM	WT File	172 KB
index-20--2259994602858926461	7/19/2015 10:43 PM	WT File	92 KB
mongod.lock	7/19/2015 10:12 PM	LOCK File	1 KB
sizeStorer	7/19/2015 10:47 PM	WT File	36 KB
storage	7/19/2015 10:12 PM	BSON File	1 KB
WiredTiger	7/19/2015 10:12 PM	File	1 KB
WiredTiger	7/19/2015 10:12 PM	BASECFG File	1 KB
WiredTiger.lock	7/19/2015 10:12 PM	LOCK File	1 KB
WiredTiger.turtle	7/19/2015 10:48 PM	TURTLE File	1 KB
WiredTiger	7/19/2015 10:48 PM	WT File	68 KB

Figure 8-11. *WiredTiger Data folder*

Separate volumes can be specified for storing indexes.

When specifying the DBPath you need to ensure that the directory corresponds to the storage engine, which is specified using the `-storageEngine` option when starting the mongod. The mongod will fail to start if the dbpath contains files created by a storage engine other than the one specified using the `-storageEngine` option. So if MMAPv1 files are found in DBPath, then WT will fail to start.

Internally, WiredTiger uses the traditional B+ tree structure for storing and managing data but that's where the similarity ends. Unlike B+ tree, it doesn't support in-place updates.

WiredTiger cache is used for any read/write operations on the data. The trees in cache are optimized for in-memory access.

Reads and Writes

You will briefly look at how the reads and writes happen. As mentioned, when MongoDB updates and reads from the DB, it is actually reading and writing to memory.

If a modification operation in the MongoDB MMAPv1 storage engine increases the record size bigger than the space allocated for it, then the entire record will be moved to a much bigger space with extra padding bytes. By default, MongoDB uses power of 2-sized allocations so that every document in MongoDB is stored in a record that contains the document itself and extra space (padding). Padding allows the document to grow as the result of updates while minimizing the likelihood of reallocations. Once the record is moved, the space that was originally occupied is freed up and will be tracked as free lists of different size. As mentioned, it's the `$freelist` namespace in the `.ns` file.

In the MMAPv1 storage engine, as objects are deleted, modified, or created, fragmentation will occur over time, which will affect the performance. The `compact` command should be executed to move the fragmented data into contiguous spaces.

Every 60 seconds the files in RAM are flushed to disk. To prevent data loss in the event of a power failure, the default is to run with journaling switched on. The behavior of journal is dependent on the configured storage engine.

The MMAPv1 journal file is flushed to disk every 100ms, and if there is power loss, it is used to bring the database back to a consistent state.

In WiredTiger, the data in the cache is stored in a B+ tree structure which is optimized for in-memory. The cache maintains an on-disk page image in association with an index, which is used to identify where the data being asked for actually resides within the page (see Figure 8-12).

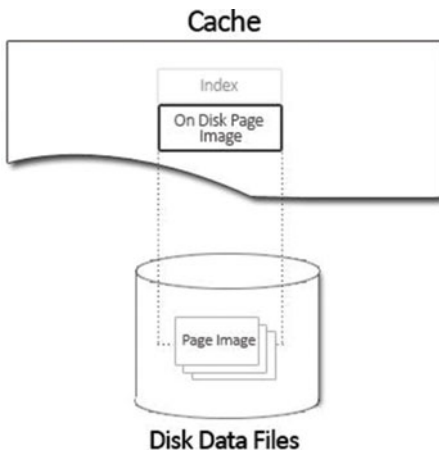


Figure 8-12. WiredTiger cache

The write operation in WiredTiger **never updates in-place**.

Whenever an operation is issued to WiredTiger, internally it's broken into **multiple transactions** wherein each transaction works within the context of an **in-memory snapshot**. The snapshot is of the committed version before the transactions started. Writers can create new versions concurrent with the readers.

The write operations do not change the **page**; instead the updates are layered on top of the page. A skipList data structure is used to maintain all the updates, where the most recent update is on the top. Thus, whenever a user reads/writes the data, the index checks whether a skiplist exists. If a skiplist is not there, it returns data from the on-disk page image. If skiplist exists, the data at the head of the list is returned to the threads, which then update the data. Once a commit is performed, the updated data is added to the head of the list and the pointers are adjusted accordingly. This way multiple users can access data concurrently without any conflict. The conflict occurs only when multiple threads are trying to update the same record. In that case, one update wins and the other concurrent update needs to retry.

Any changes to the tree structure due to the update, such as splitting the data if the page sizes increase, relocation, etc., are later reconciled by a background process. This accounts for the fast write operations of the WiredTiger engine; the task of data arrangement is left to the background process. See Figure 8-13.

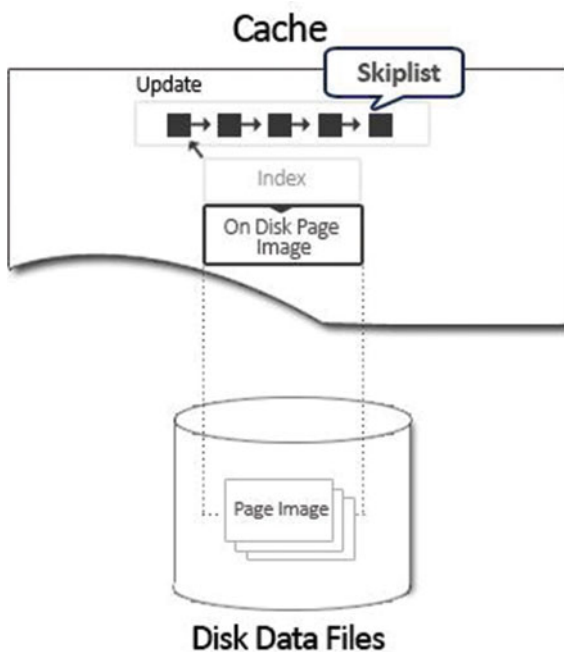


Figure 8-13. SkipList

WiredTiger uses the MVCC approach to ensure concurrency control wherein multiple versions of the data are maintained. It also ensures that every thread that is trying to access the data sees the most consistent version of the data. As you have seen, the writes are not in place; instead they are appended on top of the data in a skipList data structure with the most recent update on the top. Threads accessing the data get the latest copy, and they continue to work with that copy uninterrupted until the time they commit. Once they commit, the update is appended at the top of the list and thereafter any thread accessing the data will see that latest update.

This enables multiple threads to access the same data concurrently without any locking or contention. This also enables the writers to create new versions concurrently with the readers. The conflict occurs only when multiple threads are trying to update the same record. In that case, one update wins and the other concurrent update needs to retry.

Figure 8-14 depicts the MVCC in action.

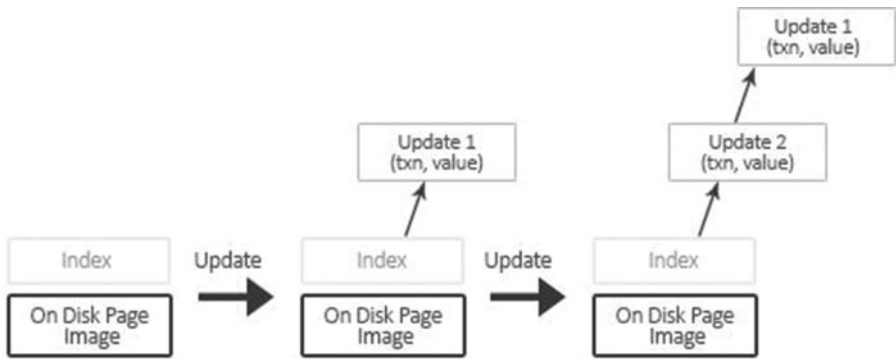


Figure 8-14. Update in action

The WiredTiger journal ensures that writes are persisted to disk between checkpoints. WiredTiger uses checkpoints to flush data to disk by default every 60 seconds or after 2GB of data has been written. Thus, by default, WiredTiger can lose up to 60 seconds of writes if running without journaling, although the risk of this loss will typically be much less if using replication for durability. The WiredTiger transaction log is not necessary to keep the data files in a consistent state in the event of an unclean shutdown, and so it is safe to run without journaling enabled, although to ensure durability the “replica safe” write concern should be configured. Another feature of the WiredTiger storage engine is the ability to compress the journal on disk, thereby reducing storage space.

How Data Is Written Using Journaling

In this section you will briefly look at how write operations are performed using journaling.

MongoDB disk writes are lazy, which means if there are 1,000 increments in one second, it will only be written once. The physical writes occurs a few seconds after the operation.

You will now see how an update actually happens in mongod.

As you know, in the MongoDB system, mongod is the primary daemon process. So the disk has the data files and the journal files. See Figure 8-15.



Figure 8-15. mongod

When the mongod is started, the data files are mapped to a **shared view**. In other words, the data file is mapped to a **virtual address space**. See Figure 8-16.

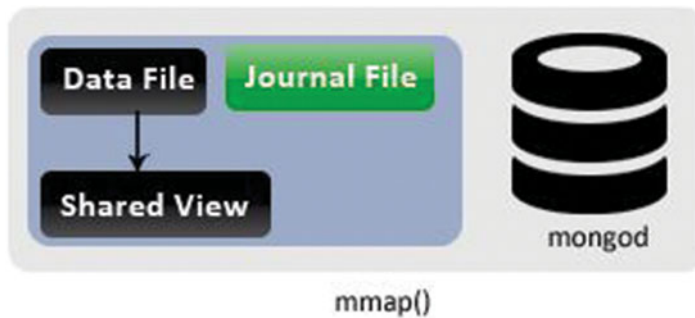


Figure 8-16. *mmap() maps to shared view*

Basically, the OS recognizes that your data file is **2000 bytes on disk**, so it maps this to memory address **1,000,000 – 1,002,000**. Note that the data will not be actually loaded **until accessed**; the OS just **maps** it and keeps it.

Until now you still had files backing up the memory. Thus any change in memory will be flushed to the underlying files by the OS.

This is how the mongod works when journaling is **not enabled**. Every **60** seconds the in-memory changes are **flushed by the OS**.

In this scenario, let's look at writes with **journaling enabled**. When journaling is enabled, a second mapping is made to a private view by the mongod.

That's why the **virtual memory amount** used by mongod **doubles** when the journaling is enabled. See Figure 8-17.

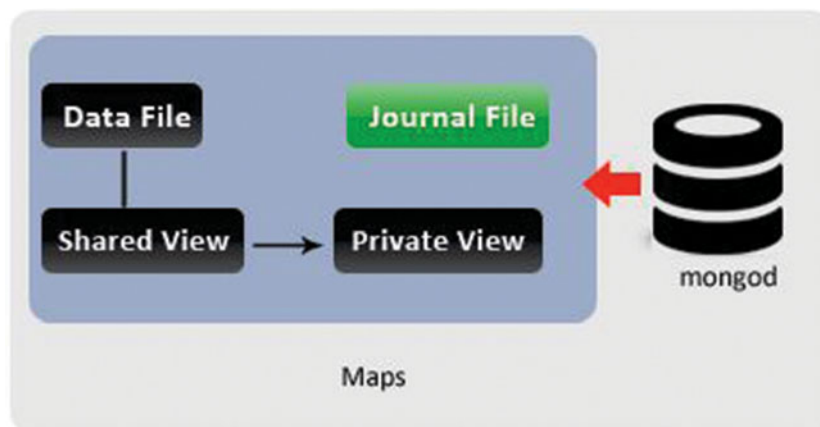


Figure 8-17. *maps to private view*

You can see in Figure 8-17 how the data file is not directly connected to the private view, so the changes will not be flushed from the private view to the disk by the OS.

Let’s see what sequence of events happens when a **write operation is initiated**. When a write operation is initiated it, **first** it writes to the private view (Figure 8-18).

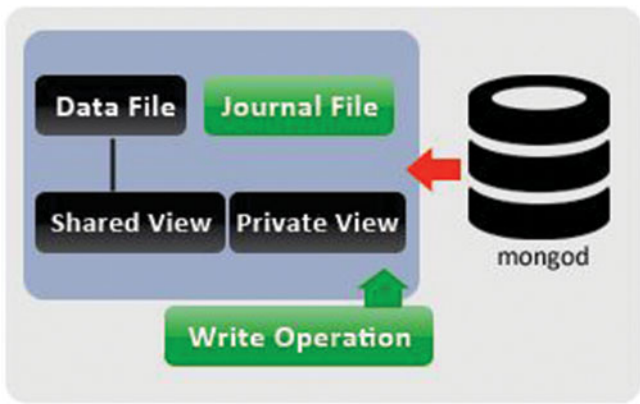


Figure 8-18. Initiated write operation

Next, the changes are written to the **journal file**, appending a **brief description** of what’s changed in the files (Figure 8-19).

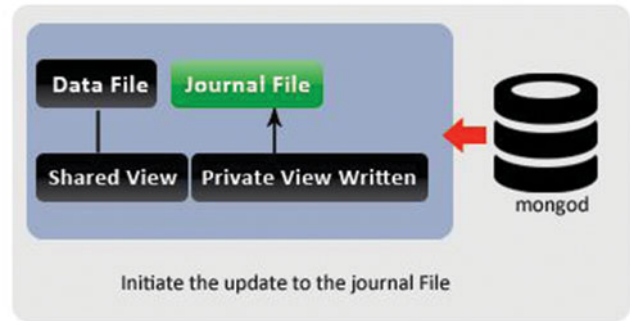


Figure 8-19. Updating the journal file

The journal keeps appending the **change description** as and when it gets the change. If the mongod fails at this point, the journal can **replay all the changes** even if the data file **is not yet modified**, making the **write safe** at this point.

The journal will now replay the logged changes on the shared view (Figure 8-20).

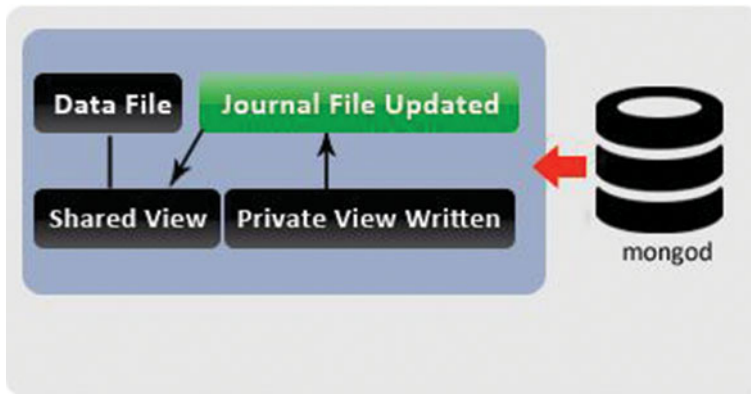


Figure 8-20. Updating the shared view

Finally, with a very fast speed the changes are written to the disk. By default, the OS is requested to do this every 60 seconds by the mongod (Figure 8-21).



Figure 8-21. Updating the data file

In the last step, the shared view is remapped to the private view by the mongod. This is done to prevent the private view from getting too dirty (Figure 8-22).

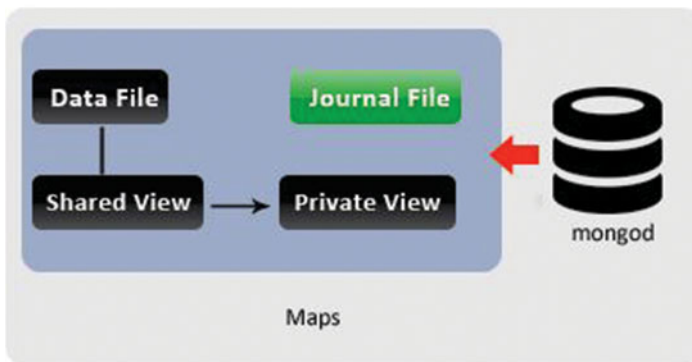


Figure 8-22. Remapping

GridFS – The MongoDB File System

You looked at what happens under the hood. You saw that MongoDB stores data in BSON documents. BSON documents have a document size limit of 16MB.

GridFS is **MongoDB's specification** for handling **large** files that exceed **BSON's document size limit**. This section will briefly cover GridFS.

Here “specification” means that it is not a MongoDB feature by itself, so there is no code in MongoDB that implements it. It just specifies how large files need to be handled. The language drivers such as PHP, Python, etc. implement this specification and expose an API to the user of that driver, enabling them to store/retrieve large files in MongoDB.

The Rationale of GridFS

By design, a MongoDB document (i.e. a BSON object) **cannot** be larger than 16MB. This is to keep performance at an **optimum level**, and the size is **well suited** for our needs. For example, 4MB of space might be **sufficient** for storing a sound clip or a profile picture. However, if the requirement is to store **high quality audio or movie clips**, or even files that are more than **several hundred megabytes** in size, MongoDB has you covered by using **GridFS**.

GridFS specifies a **mechanism** for dividing a large file among **multiple documents**. The language driver that implements it, for example, the PHP driver, takes care of the **splitting of the stored files** (or merging the split chunks when files are to be retrieved) under the hood. The developer using the driver does not need to know of **such internal details**. This way GridFS allows the developer to **store and manipulate files** in a transparent and efficient way.

GridFS uses two collections for storing the file. One collection maintains the **metadata** of the file and the other collection stores the **file's data** by breaking it into **small pieces called chunks**. This means the file is divided into smaller chunks and each chunk is stored as a separate document. By default the chunk size is limited to 255KB.

This approach not only makes the storing of data scalable and easy but also makes the range queries easier to use when a specific part of files are retrieved.

Whenever a file is queried in GridFS, the chunks are reassembled as required by the client. This also provides the user with the capability to access arbitrary sections of the files. For example, the user can directly move to the middle of a video file.

The GridFS specification is useful in cases where the file size exceeds the default 16MB limitation of MongoDB BSON document. It's also used for storing files that you need to access without loading the entire file in memory.

GridFS under the Hood

GridFS is a **lightweight specification** used for storing files.

There's no "special case" handling done at the MongoDB server for the GridFS requests. All the work is done at the client side.

GridFS enables you to **store large files** by splitting them up into **smaller chunks** and storing each of the chunks as **separate documents**. In addition to these chunks, there's one more document that contains the metadata about the file. Using this metadata information, the chunks are grouped together, forming the complete file.

The storage overhead for the chunks can be kept to a minimum, as MongoDB supports storing binary data in documents.

The two collections that are used by GridFS for storing of large files are by default named as `fs.files` and `fs.chunks`, although a different bucket name can be chosen than `fs`.

The chunks are stored by default in the `fs.chunks` collection. If required, this can be overridden. Hence all of the data is contained in the `fs.chunks` collection.

The structure of the individual documents in the chunks collection is pretty simple:

```
{
  "_id" : ObjectId("..."), "n" : 0, "data" : BinData("..."),
  "files_id" : ObjectId("...")
}
```

The chunk document has the following important keys.

- `"_id"`: This is the **unique identifier**.
- `"files_id"`: This is **unique identifier** of the document that contains the metadata related to the chunk.
- `"n"`: This is basically depicting the position of the chunk in the original file.
- `"data"`: This is the **actual binary data** that constitutes this chunk.

The `fs.files` collection stores the `metadata` for each file. Each document within this collection represents a single file in GridFS. In addition to the general metadata information, each document of the collection can contain custom metadata specific to the file it's representing.

The following are the `keys` that are `mandated` by the `GridFS specification`:

- `_id`: This is the unique identifier of the file.
- `Length`: This depicts the total bytes that make up the complete content of the file.
- `chunkSize`: This is the file's chunk size, in bytes. By default it's 255KB, but if needed this can be adjusted.
- `uploadDate`: This is the timestamp when the file was stored in GridFS.
- `md5`: This is generated on the server side and is the `md5 checksum` of the files contents. MongoDB server generates its value by using the `filemd5` command, which computes the md5 checksum of the uploaded chunks. This implies that the user can check this value to ensure that the file is uploaded correctly.

A typical `fs.files` document looks as follows (see also Figure 8-23):

```
{
  "_id" : ObjectId("..."), "length" : data_number,
  "chunkSize" : data_number,
  "uploadDate" : data_date,
  "md5" : data_string
}
```

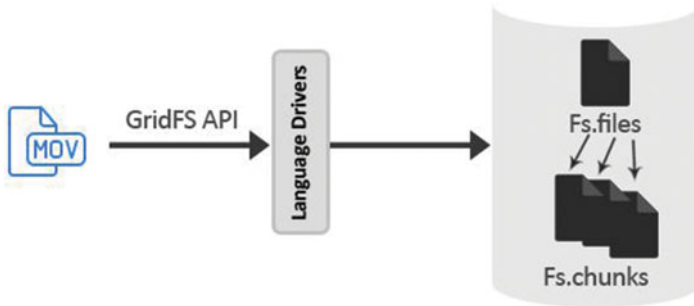


Figure 8-23. *GridFS*

Using GridFS

In this section, you will be using the PyMongo driver to see how you can start using GridFS.

Add Reference to the Filesystem

The first thing that is needed is a reference to the GridFS filesystem:

```
>>> import pymongo
>>> import gridfs
>>> myconn=pymongo.Connection()
>>> mydb=myconn.gridfstest
>>> myfs=gridfs.GridFS(db)
```

write()

Next you will execute a **basic write**:

```
>>> with myfs.new_file() as myfp:
    myfp.write('This is my new sample file. It is just grand!')
```

find()

Using the mongo shell let's see what the underlying collections holds:

```
>>> list(mydb.myfs.files.find())
[{'u'length': 38, 'u'_id': ObjectId('52fdd6189cd2fd08288d5f5c'), 'u'uploadDate': datetime.datetime(2014, 11, 04, 4, 20, 41, 800000), 'u'md5': 'u'332de5ca08b73218a8777da69293576a', 'u'chunkSize': 262144}]
>>> list(mydb.myfs.chunks.find())
[{'u'files_id': ObjectId('52fdd6189cd2fd08288d5f5c'), 'u'_id': ObjectId('52fdd6189cd2fd08288d5f5d'), 'u'data': Binary('This is my new sample file. It is just grand!', 0), 'u'n': 0}]
```

Force Split the File

Let's **force split the file**. This is done by specifying a **small chunkSize** while file creation, like so:

```
>>> with myfs.new_file(chunkSize=10) as myfp:
    myfp.write('This is second file. I am assuming it will be split into various chunks')
>>>
>>>myfp
<gridfs.grid_file.GridIn object at 0x0000000002AC79B0>
>>>myfp._id
ObjectId('52fdd76e9cd2fd08288d5f5e')
>>> list(mydb.myfs.chunks.find(dict(files_id=myfp._id)))
.....
ObjectId('52fdd76e9cd2fd08288d5f65'), 'u'data': Binary('s', 0), 'u'n': 6}]
```

read()

You now know how the file is actually stored in the database. Next, using the **client driver**, you will now read the file:

```
>>> with myfs.get(myfp._id) as myfp_read:
    print myfp_read.read()
```

"This is second file. I am assuming it will be split into various chunks."

The user need not be aware of the **chunks at all**. You need to use the APIs exposed by the client to read and write files from GridFS.

Treating GridFS More Like a File System

new_file() - Create a new file in GridFS

You can pass any number of keywords as **arguments** to the `new_file()`. This will be added in the `fs.files` document:

```
>>> with myfs.new_file(
    filename='practicalfile.txt',
    content_type='text/plain',
    my_other_attribute=42) as myfp:
    myfp.write('My New file')

>>>myfp
<gridfs.grid_file.GridIn object at 0x0000000002AC7AC8>
>>> db.myfs.files.find_one(dict(_id=myfp._id))
{'contentType': u'text/plain', u'chunkSize': 262144, u'my_other_attribute': 42,
u'filename': u'practicalfile.txt', u'length': 8, u'uploadDate': datetime.datetime(2014, 11,
04, 9, 01, 32, 800000), u'_id': ObjectId('52fdd8db9cd2fd08288d5f66'), u'md5':
u'681e10aecbafd7dd385fa51798ca0fd6'}
```

A file can be **overwritten** using filenames. Since `_id` is used for indexing files in GridFS, the old file is not removed. Just a **file version is maintained**.

```
>>> with myfs.new_file(filename='practicalfile.txt', content_type='text/plain') as myfp:
    myfp.write('Overwriting the "My New file"')
```

get_version()/get_last_version()

In the above case, `get_version` or `get_last_version` can be used to retrieve the file with the filename.

```
>>>myfs.get_last_version('practicalfile.txt').read()
'Overwriting the "My New file"'
>>>myfs.get_version('practicalfile.txt',0).read()
'My New file'
```

You can also list the files in GridFS:

```
>>>myfs.list()
[u'practicalfile.txt', u'practicalfile2.txt']
```

delete()

Files can also be removed:

```
>>>myfp=myfs.get_last_version('practicalfile.txt')
>>>myfs.delete(myfp._id)
>>>myfs.list()
[u'practicalfile.txt', u'practicalfile2.txt']
>>>myfs.get_last_version('practicalfile.txt').read()
'My New file'
>>>
```


Note that only one version of `practicalfile.txt` was removed. You still have a file named `practicalfile.txt` in the filesystem.

`exists()` and `put()`

Next, you will use `exists()` to check if a file exists and `put()` to quickly write a short file into GridFS:

```
>>>myfs.exists(myfp._id)
False
>>>myfs.exists(filename='practicalfile.txt')
True
>>>myfs.exists({'filename':'practicalfile.txt'}) # equivalent to above
True
>>>myfs.put('The red fish', filename='typingtest.txt')
ObjectId('52fddbc69cd2fd08288d5f6a')
>>>myfs.get_last_version('typingtest.txt').read()
'The red fish'
>>>
```

Indexing

In this part of the book, you will briefly examine what an index is in the MongoDB context. Following that, we will highlight the various types of indexes available in MongoDB, concluding the section by highlighting the behavior and limitations.

An index is a data structure that speeds up the read operations. In layman terms, it is comparable to a book index where you can reach any chapter by looking in the index for the chapter and jumping directly to the page number rather than scanning the entire book to reach to the chapter, which would be the case if no index existed.

Similarly, an index is defined on fields, which can help in searching for information in a better and efficient manner.

As in other databases, in MongoDB also it's perceived in a similar fashion (it's used for speeding up the `find()` operation). The type of queries you run help to create efficient indexes for the databases. For example, if most of the queries use a Date field, it would be beneficial to create an index on the Date field. It can be tricky to figure out which index is optimal for your query, but it's worth a try because the queries that otherwise take minutes will return results instantaneously if a proper index is in place.

In MongoDB, an index can be created on any field or sub-field of a document. Before you look at the various types of indexes that can be created in MongoDB, let's list a few core features of the indexes:

- The indexes are defined at the per-collection level. For each collection, there are different sets of indexes.
- Like SQL indexes, a MongoDB index can also be created either on a single field or set of fields.
- In SQL, although indexes enhance the query performance, you incur overhead for every write operation. So before creating any index, consider the type of queries, frequency, the size of the workload, and the insert load along with application requirements.
- A BTree data structure is used by all MongoDB indexes.
- Every query using the update operations uses only one index, which is decided by the query optimizer. This can be overridden by using a hint.

- A query is said to be covered by an index if **all fields** are part of the **index**, irrespective of whether it's used for querying or for projecting.
- A **covering index** maximizes the **MongoDB performance** and **throughput** because the query can be **satiated** using an index only, without **loading the full documents** in memory.
- An index will only be **updated** when the **fields** on which the index has been created are changed. Not all **update operations** on a document cause the index to be changed. It will only be changed **if the associated fields are impacted**.

Types of Indexes

In this section, you will look at the different types of indexes that are available in MongoDB.

_id index

This is the default index that is created on the `_id` field. This index cannot be deleted.

Secondary Indexes

All indexes that are user created using `ensureIndex()` in MongoDB are termed as secondary indexes.

1. These indexes can be created on any field in the document or the sub document. Let's consider the following document:

```
{ "_id": ObjectId(...), "name": "Practical User", "address":
  { "zipcode": 201301, "state": "UP" } }
```

In this document, an index can be created on the name field as well as the state field.

2. These indexes can be created on a field that is holding a sub-document.

If you consider the above document where address is holding a sub-document, in that case an index can be created on the address field as well.

3. These indexes can either be created on a single field or a set of fields. When created with set of fields, it's also termed a **compound index**.

To explain it a bit further, let's consider a products collection that holds documents of the following format:

```
{ "_id": ObjectId(...), "category": ["food", "grocery"], "item": "Apple",
  "location": "16th Floor Store", "arrival": Date(...) }
```

If the maximum of the queries use the fields Item and Location, then the following compound index can be created:

```
db.products.ensureIndex ({"item": 1, "location": 1})
```

In addition to the query that is referring to all the fields of the compound index, the above compound index can also support queries that are using any of the index prefixes (i.e. it can also support queries that are using only the item field).

4. If the index is created on a field that holds an array as its value, then a multikey index is used for indexing each value of the array separately.

Consider the following document:

```
{ "_id" : ObjectId("..."), "tags" : [ "food", "hot", "pizza", "may" ] }
```

An index on tags is a multikey index, and it will have the following entries:

```
{ tags: "food" }
{ tags: "hot" }
{ tags: "pizza" }
{ tags: "may" }
```

5. Multikey compound indexes can also be created. However, at any point, only one field of the compound index can be of the array type.

If you create a compound index of {a1: 1, b1: 1}, the permissible documents are as follows:

```
{a1: [1, 2], b1: 1}
{a1: 1, b1: [1, 2]}
```

The following document is not permissible; in fact, MongoDB won't be even able to insert this document:

```
{a1: [21, 22], b1: [11, 12]}
```

If an attempt is made to insert such a document, the insertion will be rejected and the following error results will be produced: "cannot index parallel arrays".

You will next look at the various options/properties that might be useful while creating indexes.

Indexes with Keys Ordering

MongoDB indexes maintain references to the fields. The references are maintained in either an ascending order or descending order. This is done by specifying a number with the key when creating an index. This number indicates the index direction. Possible options are 1 and -1, where 1 stands for ascending and -1 stands for descending.

In a single key index, it might not be too important; however, the direction is very important in compound indexes.

Consider an Events collection that includes both username and timestamp. Your query is to return events ordered by username first and then with the most recent event first. The following index will be used:

```
db.events.ensureIndex({ "username" : 1, "timestamp" : -1 })
```

This index contains references to the documents that are sorted in the following manner:

1. First by the username field in ascending order.
2. Then for each username sorted by the timestamp field in the descending order.

Unique Indexes

When you create an index, you need to ensure uniqueness of the values being stored in the indexed field. In such cases, you can create indexes with the Unique property set to true (by default it's false).

Say you want a unique_index on the field `userid`. The following command can be run to create the unique index:

```
db.payroll.ensureIndex( { "userid": 1 }, { unique: true } )
```

This command ensures that you have unique values in the `user_id` field. A few points that you need to note for the uniqueness constraint are

- If the unique constraint is used on a compound index in that scenario, uniqueness is enforced on the combination of values.
- A null value is stored in case there's no value specified for the field of a unique index.
- At any point only one document is permitted without a unique value.

dropDups

If you are creating a unique index on a collection that already has documents, the creation might fail because you may have some documents that contain duplicate values in the indexed field. In such scenarios, the `dropDups` options can be used for force creation of the unique index. This works by keeping the first occurrence of the key value and deleting all the subsequent values. By default `dropDups` is false.

Sparse Indexes

A sparse index is an index that holds entries of the documents within a collection that has the fields on which the index is created. If you want to create a sparse index on the `LastName` field of the `User` collection, the following command can be issued:

```
db.User.ensureIndex( { "LastName": 1 }, { sparse: true } )
```

This index will contain documents such as

```
{FirstName: Test, LastName: User}
or
{FirstName: Test2, LastName: }
```

However, the following document will not be part of the sparse index:

```
{FirstName: Test1}
```

The index is said to be sparse because this only contains documents with the indexes field and miss the documents when the fields are missing. Due to this nature, sparse indexes provide a significant space saving.

In contrast, the non-sparse index includes all documents irrespective of whether the indexed field is available in the document or not. Null value is stored in case the fields are missing.

TTL Indexes (Time To Live)

A new index property was introduced in version 2.2 that enables you to remove documents from the collection automatically after the specified time period is elapsed. This property is ideal for scenarios such as logs, session information, and machine-generated event data, where the data needs to be persistent only for a limited period.

If you want to set the TTL of one hour on collection logs, the following command can be used:

```
db.Logs.ensureIndex( { "Sample_Time": 1 }, { expireAfterSeconds: 3600 } )
```

However, you need to note the following limitations:

- The field on which the index is created must be of the date type only. In the above example, the field `sample_time` must hold date values.
- It does not support compound indexes.
- If the field that is indexed contains an array with multiple dates, the document expires when the smallest date in the array matches the expiration threshold.
- It cannot be created on the field which already has an index created.
- This index cannot be created on capped collections.
- TTL indexes expire data using a background task, which is run every minute, to remove the expired documents. So you cannot guarantee that the expired document no longer exists in the collection.

Geospatial Indexes

With the rise of the smartphone, it's becoming very common to query for things near a current location. In order to support such location-based queries, MongoDB provides geospatial indexes.

To create a geospatial index, a coordinate pair in the following forms must exist in the documents:

- Either an array with two elements
- Or an embedded document with two keys (the key names can be anything).

The following are valid examples:

```
{ "userloc" : [ 0, 90 ] }
{ "loc" : { "x" : 30, "y" : -30 } }
{ "loc" : { "latitude" : -30, "longitude" : 180 } }
{ "loc" : { "a1" : 0, "b1" : 1 } }.
```

The following can be used to create a geospatial index on the `userloc` field:

```
db.userplaces.ensureIndex( { userloc : "2d" } )
```

A geospatial index assumes that the values will range from -180 to 180 by default. If this needs to be changed, it can be specified along with `ensureIndex` as follows:

```
db.userplaces.ensureIndex({ "userloc" : "2d"}, {"min" : -1000, "max" : 1000})
```

Any documents with values beyond the maximum and the minimum values will be rejected. You can also create compound geospatial indexes.

Let's understand with an example how this index works. Say you have documents that are of the following type:

```
{ "loc": [0, 100], "desc": "coffeeshop" }
{ "loc": [0, 1], "desc": "pizzashop" }
```

If the query of a user is to find all coffee shops near her location, the following compound index can help:

```
db.ensureIndex({ "userloc" : "2d", "desc" : 1 })
```

Geohaystack Indexes

Geohaystack indexes are bucket-based geospatial indexes (also called **geospatial haystack indexes**). They are useful for queries that need to find out locations in a small area and also need to be filtered along another dimension, such as finding documents with coordinates within 10 miles and a type field value as restaurant.

While defining the index, it's mandatory to specify the `bucketSize` parameter as it determines the haystack index granularity. For example,

```
db.userplaces.ensureIndex({ userpos : "geoHaystack", type : 1 }, { bucketSize : 1 })
```

This example creates an index wherein keys within 1 unit of latitude or longitude are stored together in the same bucket. You can also include an additional category in the index, which means that information will be looked up at the same time as finding the location details.

If your use case typically searches for "nearby" locations (i.e. "restaurants within 25 miles"), a haystack index can be more efficient.

The matches for the additional indexed field (e.g. category) can be found and counted within each bucket.

If, instead, you are searching for "nearest restaurant" and would like to return results regardless of distance, a normal 2d index will be more efficient.

There are currently (as of MongoDB 2.2.0) a few limitations on haystack indexes:

- Only one additional field can be included in the haystack index.
- The additional index field has to be a single value, not an array.
- Null long/lat values are not supported.

In addition to the above mentioned types, there is a new type of index introduced in version 2.4 that supports text search on a collection.

Previously in beta, in the 2.6 release, text search is a built-in feature. It includes options such as searching in 15 languages and an aggregation option that can be used to set up faceted navigation by product or color, for example, on an e-commerce website.

Index Intersection

Index intersection is introduced in version 2.6 wherein multiple indexes can be intersected to satiate a query. To explain it a bit further, let's consider a products collection that holds documents of the following format

```
{ "_id": ObjectId(...), "category": ["food", "grocery"], "item": "Apple", "location": "16th Floor Store", "arrival": Date(...) }.
```

Let's further assume that this collection has the following two indexes:

```
{ "item": 1 }.
{ "location": 1 }.
```

Intersection of the above two indexes can be used for the following query:

```
db.products.find ({ "item": "xyz", "location": "abc" })
```

You can run `explain()` to determine if index intersection is used for the above query. The `explain` output will include either of the following stages: `AND_SORTED` or `AND_HASH`. When doing index intersection, either the entire index or only the index prefix can be used.

You next need to understand how this index intersection feature impacts the compound index creation.

While creating a compound index, both the order in which the keys are listed in the index and the sort order (ascending and descending) matters. Thus a compound index may not support a query that does not have the index prefix or has keys with different sort order.

To explain it a bit further, let's consider a `products` collection that has the following compound index:

```
db.products.ensureIndex ({ "item": 1, "location": 1 })
```

In addition to the query, which is referring to all the fields of the compound index, the above compound index can also support queries that are using any of the index prefix (it can also support queries that are using only the `item` field). But it won't be able to support queries that are using either only the `location` field or are using the `item` key with a different sort order.

Instead, if you create two separate indexes, one on the `item` and the other on the `location`, these two indexes can either individually or through intersections support the four queries mentioned above. Thus, the choice between whether to create a compound index or to rely on intersection of indexes depends on the system's needs.

Note that index intersection will not apply when the `sort()` operation needs an index that is completely separate from the query predicate.

For example, let's assume for the `products` collection you have the following indexes:

```
{ "item": 1 }.
{ "location": 1 }.
{ "location": 1, "arrival_date": -1 }.
{ "arrival_date": -1 }.
```

Index intersection will not be used for the following query:

```
db.products.find( { item: "xyz" } ).sort( { location: 1 } )
```

That is, MongoDB will not use the `{ item: 1 }` index for the query, and the separate `{ location: 1 }` or the `{ location: 1, arrival_date: -1 }` index for the sort.

However, index intersection can be used for the following query since the index `{ location: 1, arrival_date: -1 }` can fulfil part of the query predicate:

```
db.products.find( { item: { "xyz" } , location: "A" } ).sort( { arrival_date: -1 } )
```

Behaviors and Limitations

Finally, the following are a few behaviors and limitations that you need to be aware of:

- More than [64 indexes](#) may not be allowed in a collection.
- Index keys cannot be larger than [1024 bytes](#).
- A document cannot be indexed if its fields' values are greater than this size.
- The following command can be used to query documents that are too large to index:

```
db.practicalCollection.find({<key>: <too large to index>}).hint({$natural: 1})
```

- An index name (including the [namespace](#)) must be less than [128 characters](#).
- The insert/update speeds are impacted to some extent by an index.
- Do not maintain indexes that are not used or will not be used.
- Since each clause of an [\\$or](#) query executes in parallel, each can use a different index.
- The queries that use the [sort\(\)](#) method and the [\\$or](#) operator will not be able to use the indexes on the [\\$or](#) fields.
- Queries that use the [\\$or](#) operator are not supported by the second [geospatial query](#).

Summary

In this chapter, you covered how data is stored under the hood and how writes happen using journaling. You also looked at GridFS and the different types of indexes available in MongoDB.

In the following chapter, you will look at MongoDB from administration perspective.